A Context-Based Behavioral Language for IoT

Achiya Elyasaf[1], Assaf Marron[2], Arnon Sturm[1], and Gera Weiss[1]

[1] Ben-Gurion University of the Negev {achiya,sturm,geraw}@bgu.ac.il
[2] Weizmann Institute of Science assaf.marron@weizmann.ac.il

Abstract. As devices, platforms, and technologies for IoT (Internet-of-Things) and robots, develop, the question of how to best specify the behavior of such systems so that it is both robust and manageable becomes central. Current practices may suffice when working with simple requirements. However, behavior specification given in current languages often become unwieldy as they grow to accommodate complex conditions, exceptions, and priorities. To address this, we propose to use the scenario-based programming approach, and specifically, the graphical language of live sequence charts (LSC). This addresses one aspect of the specification growth issue by allowing a natural break-down of the specification in alignment with the requirements. The other aspect of our solution, aiming at further simplifying and shortening the specification, is based on subjecting these scenarios to context—a key concept in IoT and autonomous robot modeling. Specifically, we propose additions to LSC for subjecting behavioral scenario charts to contexts and a methodology to work with these idioms.

1 Introduction

The *Internet of Things* (IoT) is gaining attention from both industry and academia. In its core, the IoT technology allows "people and things to be connected Anytime, Anyplace, with Anything and Anyone, ideally using Any path/network and Any service" [12]. The primary goal is to create "a better world for human beings", where devices in our living environment are programmed to know what we want and what we need and to act accordingly without direct instructions [10]. While many current IoT systems serve mainly for sensing, data analysis and reporting [2], the focus of this paper is on reactivity, i.e., on systems whose behavior is focused on reactions like robots, smart-building automation, traffic-light management, etc.

Bill Wasik [14] pointed out that IoT is making our world "programmable", outlining three phases in this direction: (1) getting more objects onto the network; (2) programming their actions to work automatically; and (3) understanding connected objects as a single system to be programmed and creating complex relationships among them. In view of the current advances in the IoT technologies [2] it seems that we are somewhere between the first and the second phases.

A key ingredient in rich IoT interactions is *context*. Contexts can be defined as information that can be used to characterize the situation of entities or processes in a system [1]. The term *context-awareness* refers to the system's ability to use context information [1]. Various studies related to context-aware systems and programming architectures have already shown the importance of context for IoT [10, 9]. The European Union has identified context awareness as an important IoT research area and specified a time frame (2015–2020) for context-awareness research and development [13].

Towards the third phase in Wasik's description of IoT evolution, we propose in this paper to use rich specification languages that allow scenario-based (multi-step) specifications. Using a language that is capable of modeling complex behaviors as a composition of stand-alone scenarios, each of which introduces multiple actions and triggers, as well as constraints that affect other scenarios, we aim at more natural specifications that better capture the scenarios that users want to express.

The specification language that we propose is based on the LSC language [6, 3]. The language is an extension of message sequence charts (MSC) with the ability to specify possible and mandatory behaviors of reactive systems [8], as well as forbidden behavior. Due to its graphical representation and tool support [5], LSC makes for an excellent candidate for a modeling language for IoT. However, while the core language readily answers the requirement of allowing direct specification and composition of scenarios with multiple sensing and actuations, it lacks a direct support for context. Since, as said above, context is a central notion in IoT, and particularly, in specifying complex behaviors of integrated IoT systems, we propose an extension of the LSC language with idioms for explicitly defining contexts and referencing them.

2 Requirements from a Modeling Language

Following IoT systems' characteristics by Perera [10, 9], we set the requirements from a modeling language (abbr. ML) that would facilitate their configuration and specification. These requirements are divided into three viewpoints as follows:
The behavior/business process viewpoint. The ML should allow specification of:

RQ1 *data/objects*, used by the system and the IoT devices.
RQ2 acting upon historical data (stateful), e.g., for reacting to sequences of events.
RQ3 *desired* behaviors—the system core that aim at supporting various scenarios.
RQ4 *undesired* behaviors—to specify explicitly behaviors that the system implicitly avoids. This allows for catching mistakes as a conflict between a negative behavioral requirement and a future specification or a code artifact.
RQ5 *contexts*, e.g., location, time, a sequence of events, or a particular system state. Since the behavior of IoT systems is inherently context-dependent, the context should play a major role within the specification.
RQ6 run-time *adaptation policy* that includes conflict resolution and priorities.

The potential users viewpoint. The ML should:

RQ7 be *easily explained*, as it aims for engineers and end-users [4].
RQ8 have *automated usage guidance* (e.g., templates and heuristics) to guide the behavior specification. To handle the system's complexity by users with various skills.

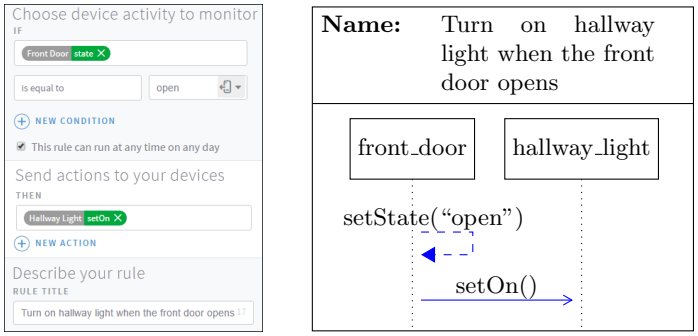
The software engineering viewpoint. The ML should:

RQ9 be modular and allow for *incremental specification*, as the requirements are not usually known in advance and changes would be continuously introduced.
RQ10 have *formal semantics* for simulation, formal analysis, optimization, etc.
RQ11 allow the specification of a *generic functionality/behavior*, as there are many devices (or their software counterparts) with similar behavior.
RQ12 support weaving of cross-cutting concerns as security, and error handing.

The LSC language on which we base this paper already supports most of these requirements. The rest are covered by the extension we discuss in the next section.

3 Contextual LSC

In this section we present *Contextual Live Sequence Charts* (Contextual LSC), an extension to the *Live Sequence Charts* (LSC) language. We explore the syntax through an example and discuss the semantics later on. We follow an example inspired by Samsung's *ARTIK-Cloud* introduction video [3] that takes place in a smart home system. ARTIK-Cloud [11, 15] is an open data exchange platform by Samsung designed to connect devices and applications. The framework allows users to specify behaviors of their devices via simple rules. A rule example is "If a door sensor detects that it is open, then turn on the light". The rule condition in ARTIK-Cloud may refer to any physical device connected to the system, but not to logical entities or contexts.



(a) The rule in ARTIK (b) The scenario in LSC

Fig. 1. Interaction between a sensor and a light, specified in ARTIK Cloud and LSC.

3.1 Live Sequence Charts

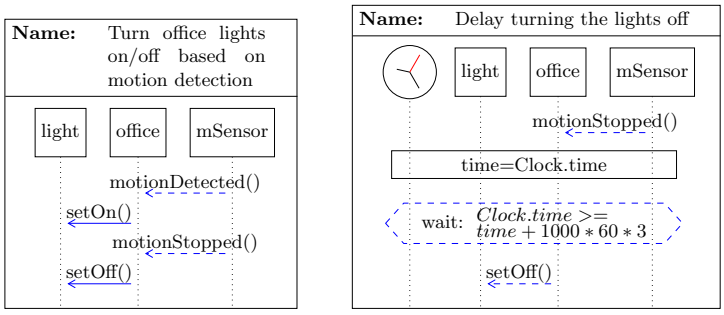
Live Sequence Charts (LSC) [6] is a diagrammatic *Scenario-Based Programming* (SBP) language that extends the language of message sequence charts (MSC) mainly through addition of modalities and by providing executable semantics. The language centers on natural and incremental specification of behaviors. It allows for modeling and coding applications as multi-modal scenarios, each corresponding to an individual requirement. The scenarios specify what can, must, or may not happen following certain sequences of events and/or when certain conditions hold. The behavior specification capability addresses RQ3 and RQ4. The incrementality of specification addresses RQ9.

In this paper we refer to the PlayGo [5] implementation of the LSC language, which has an execution engine for scenarios (or charts). This addresses RQ10.

The LSC Terminology. To get familiar with the LSC language let us start with the smart home example. We wish to turn the lights on upon opening the door. Fig. 1 depicts the rule definition in ARTIK Cloud and the corresponding scenario in LSC.

In LSC, the vertical lines, termed *lifelines*, represent instances of system classes, and horizontal lines represent messages (events, or method invocations) between system instances. Dashed and solid message lines represent monitored and execution messages, respectively. In our chart, for example, the scenario begins with a monitored message represented by a dashed arrow. Once it is triggered (in our example, the door sets its own state into "open", in a lower software level), the execution engine advances the program

[3] https://www.youtube.com/watch?v=J9vhdt5jXf0



(a) Turn the lights on/off (b) Anti-scenario: Delay setOff()

Fig. 2. Incrementality and anti-scenarios.

(from the top of the chart to its bottom). The next message is a solid arrow, representing an execution message. The engine will call the corresponding method (when it is not forbidden by another scenario). Here, the setOn method of the hallway_light object is called. Prioritizing events and smart event selection are possible as well (addressing RQ6), however due to lack of space, we will not elaborate on all the language features. The reader is referred to Harel and Marelly [6] for more information.

Incrementality and Alignment With the Requirements. Our next example demonstrates how each requirement of a system can be modelled in a separate module, allowing for the two main features of LSC—*incremental development* and *natural alignment with the requirements* (see [7]. Suppose that we wish to turn on the office lights whenever there is a person in the room. For that, we install a motion sensor and add a scenario that turn the lights on once a motion is detected, waits for the motion to stop and then turns the lights off (Figure 2a).

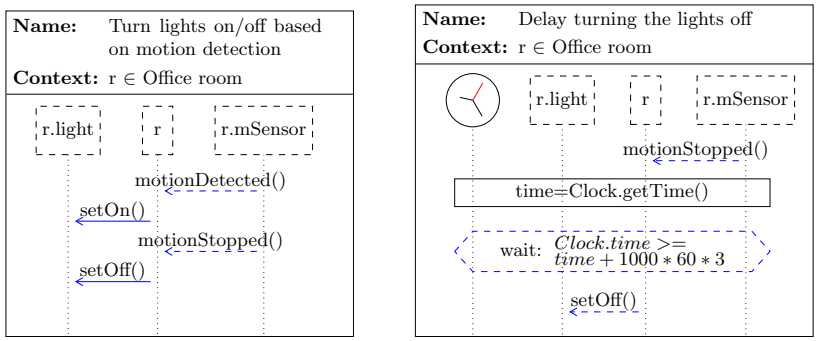
Unfortunately, after the system was deployed, a flaw was detected as the light turned off when a person did not move inside the room. While we can add a condition before calling setOff, the scenario-based paradigm encourages us to add a new scenario for each new requirement. In this case, we add an *anti-scenario*, depicted in Fig. 2b. Once a motionStopped call is monitored, the scenario collects the current time, waits for three minutes and then waits for the setOff event to take place. If the setOff method is called before the three minutes pass—it will violate this scenario and the execution engine will not choose to execute the setOff method. The scenario in Fig. 2b thus forbids the turning off of the light for three minutes. This also addresses RQ4.

3.2 Subjecting charts to contexts

As described in Section 1, a powerful context management facility is an essential tool for modeling IoT. In this section we propose an extension of the LSC syntax with idioms for handling contexts. Specifically, we propose to handle contextual data in terms of a relational data model, addressing RQ1 and RQ5.

In typical IoT applications, it is often required to subject a behavior to a context. We propose to do this by an automatic instantiation of an LSC chart, that serves as a template, whenever an instance of the context arises. Specifically, we propose semantics that allow to associate scenarios with a query over the context data. Using the standard dynamic binding semantics of LSC, we show in Section 4 how a new instance (called 'live copy' in the LCS literature) is created to handle each instance of an answer to the query. We then say that the chart is "subjected" to a context and

specifies a behavior that is attached to this context. We propose that the 'select' queries and the 'update' commands will be managed in a separated 'query and command repository' that serves as an abstraction layer between the data-model and the charts (i.e., behavioral specification), giving names to contexts that can be used by charts.



(a) Turn the lights on/off (b) Anti-scenario: Delay turning lights off

Fig. 3. Office-room Context

The concept describe in the above paragraph is best explained by continuing our example, as follows. We now refer to an office building with several office rooms that have the same setup of a smart light and a motion sensor. We wish to define this setup as a template, and set predefined scenarios for it (i.e., the scenarios in Fig. 2). Moreover, consider a smart building with various room types (e.g., as office, bedroom, kitchen, etc.), where all rooms of the same type—have a set of baseline scenarios. To this end we add a data type called Room to the context data model, to be used as a glue for smart home devices in one room. To the query and command repository we add the "Office room" query, defined as "r ∈ Room: r.type=Office" (assuming we add a 'type' attribute to Room). Each of our scenarios can then be bound to the query results, thus adding room-type-based scenarios. Fig. 3 replaces the office lifelines of Fig. 2 with r ∈ Office room. The smart light and the motion sensor are now members of r, as well as the room type. There are two important syntax changes: 1) The dashed lifeline frame denotes that the instance identity is unknown during compile time, thus the binding is done dynamically during runtime (we bind r to each instance of the context, i.e., we apply the universal binding rule of LSC); and 2) The chart properties are now its name and the query that it must bound to prior its execution. In our example, the binding to r succeeds if and only if there is an instance of Room with the type of Office. We say that the chart is bound to a context query, and more specifically that it is subjected to the 'Office room' context. The explicit introduction of context to the language addresses RQ5. and RQ12, whereas its weaving to the actual behavior addresses RQ11.

In some cases we wish to reflect modes or states of our system (e.g., emergency mode, power-saving mode, etc.). Consider, for example, a case of an emergency where we wish to temporarily keep all lights on, until the emergency situation ends. Fig. 4 handles this new requirement by adding a new context data type called Emergency, and binding an anti-scenario to it. Even though the scenario does not include the Emergency lifeline, the entire scenario will be triggered only if both queries are satisfied, i.e., there exist both Emergency (a singleton in this case) and Room instances.

Sometimes, the chart needs to be bound to a dynamic environmental changes (e.g., someone is in the room, there is a meeting in this room, etc.). Consider, for example, the lights scenarios in Fig. 3. The motion sensor signaled the room that there is a motion

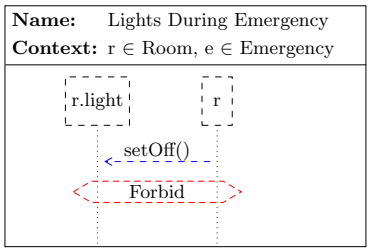


Fig. 4. Emergency Mode. Binding the entire scenario to the Emergency context. The red forbid element denotes that reaching it will violate the entire program execution.

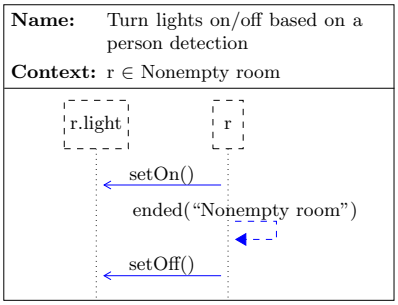
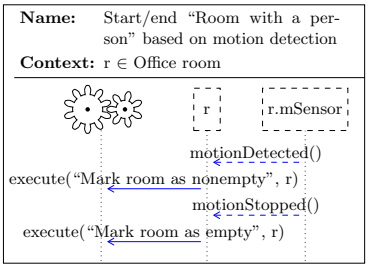


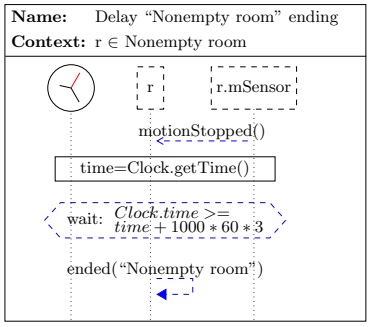
Fig. 5. Binding charts to dynamically created context. There is no need for the anti-scenario as it is abstracted.

and the room deduced that there is someone in the room. However, there might be other ways to detect a person such as a camera or a microphone or a combination thereof. Thus, the detection that there is someone in the room should be abstracted to new scenarios (described below) that will execute 'update' commands to the context data. These commands will dynamically start and end a context called 'Nonempty room' (i.e., a new answer will be generated for this query). The light scenario is now cleaner, as depicted in Fig. 5. To detect the end of the 'Nonempty room' context and set the lights off, a special predefined event, 'ended(<context name>)', is monitored.

3.3 Starting and Ending Contexts



(a) Execute commands to Start/end context(s).



(b) Delay the context ending

Fig. 6. Context activation and termination

Fig. 6 demonstrates how to start and end the 'Nonempty room' context. In the leftmost scenario, the gears lifeline represents the 'query and command repository'. Once a motion is detected, we execute an 'update' command called 'Mark room as nonempty' with r, the office room. The change to the context data triggers a new result to the 'Nonempty room', thus generating a new live copy of the scenario presented in Fig. 5. Similarly, the context data changes caused by the execution of the 'Mark room as empty' command, triggers the ended('Nonempty room') event (explained in Section 4).

4 Translational Semantics Definition

We propose a translation from contextual LSC to standard LSC. This translation provides formal semantics (RQ10) as it transforms charts that apply the new idioms to charts that only use idioms with formally specified semantics. At the core of the translation is an embedded database system that runs within the execution engine. The execution of LSC specifications together with external systems is well defined (using the super-step semantics of LSC [6]). The semantics of database systems is also well defined. Thus, the combination of the two gives the ability to give with exact execution semantics for storing context and for relating charts to it.

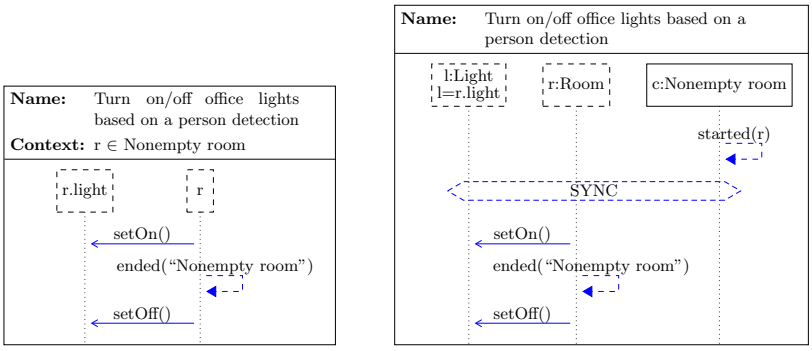
We suppose that we have a (real-time) database system for maintaining context data. We assume that context 'select' queries can be automatically translated to database views that allow for triggering a stored procedure whenever a record is added or removed from a view. We also assume that context 'update' commands can be translated to stored procedures that, when invoked, update the database as required (by adding, deleting, and changing objects and object relations).

With these assumptions, standard LSC semantics allows for triggering an event whenever a new answer to a query becomes available. This enables us to translate contextual charts, i.e., ones that are subjected to context, as follows. For every context query the standard LSC contains a dynamically-bound lifeline (this lifeline is conveniently hidden in the original contextual LSC). The standard LSC chart then starts with monitoring the event of creating this context object, and proceeds to synchronize with all other objects in the standard chart. The message that monitors the context-creation event points to the object that the query returns (like r in the phrase "r $\in$ Nonempty office room") as an unbound parameter. Once the event is triggered (when a new view-record appears), the chart variable is bound automatically. When a chart is subjected to multiple context queries (like "r $\in$ Nonempty office room, e $\in$ Emergency") its translation monitors all view-record-added events. The view-records may be created in any order, i.e., any triggering order of the monitored events needs to be detected. We thus allocated in the standard LSC a separate object with a separate lifeline per view. The context object also has a record-removed method, allowing for charts to explicitly monitor context completion. Fig. 7 depicts the translation from our contextual LSC syntax to the PlayGo LSC syntax. Since we wanted this lifeline to be hidden from the programmer, the translation also includes a chart which that is hidden altogether that monitors the record-removed event, and generates an 'ended(<context name>)' event. This chart propagates the record-removed method to the involved objects. For example, The 'ended' message in Fig. 5 is this propagated message.

As shown in Fig. 6a, we also propose to allow for using the query and command repository as an explicit lifeline represented by the gears symbol. The lifeline is translated to a regular LSC lifeline with an execute method that modifies the database.

5 A Methodology

In this section, towards a partial answer to RQ8, we elaborate on a methodology for using the proposed language for configuration and customization of IoT installation. We describe the methodology using a running example of specifying a smart-building system. We first list a set of requirements and introduce the methodology and demonstrate it via these requirements. Next, we introduce additional requirements and refine the system specification, demonstrating the flexibility and agility offered by the methodology.



(a) Contextual LSC (b) Translation to PlayGo.

Fig. 7. Semantics: the translation from a Contextual LSC to PlayGo LSC

The initial requirements are as follows:
Physical: **R1**) The room types include offices, kitchens, and restrooms; **R2**) Each room has a motion detector and a smart light; **R3**) An Office has a smart air-conditioner; **R4**) Events are emitted when a motion starts or stops.
Behavioral: **R5**) In all rooms, the light should be turned on once a motion is detected, and should be turned off if there is no motion detection for three minutes; **R6**) In office rooms, the air-conditioner should be turned on once a motion is detected, and should be turned off if there is no motion detection for three minutes; **R7**). In emergency, lights that are on must not be turned off.

Our proposed methodology is depicted in Fig. 8. It consists of four steps that can be repeated iteratively, where the order of the steps can emerge as the development progresses. In Fig. 8 the steps are indicated as ellipses whereas the artifacts are presented as rectangles. In the following we elaborate on the various steps and artifacts.

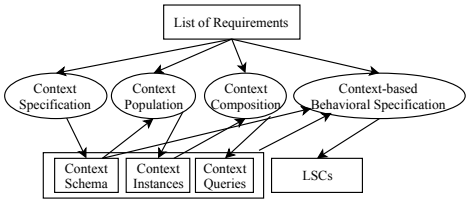


Fig. 8. Artifacts (boxes), activities (ovals), and dependencies (arrows).

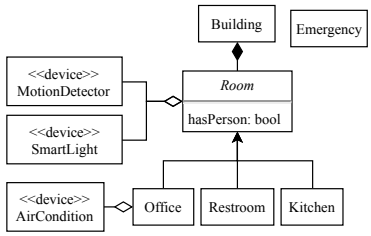


Fig. 9. The context schema

Context Specification Following the context-oriented notion we propose in this work, we start the development with specifying the contexts in mind, in light of the given requirements. In our view, everything may be included in as context data, ranging from physical entities to logical ones. A context data type might be characterized with attributes, specifying what kind of data is needed for operating the system (addressing RQ1 and RQ2). For example, in the case of the smart building, we may define the following data types: originating from R1: Building, Room, Kitchen, Office, Restroom; originating from R5 and R6: the hasPerson attribute of Room; originating from R7: Emergency. We further define the devices, i.e., MotionDetector, SmartLight (R2), and AirCondition (R3). We then define the relationships among the data types and the devices. Of course this is just one of the ways to define the schema for these require-

ID	Req.	Name	Query
Q1	R6	Office room	r ∈ Office
Q2	R5, R7	Room	r ∈ Room
Q3	R7	Emergency	e ∈ Emergency
Q4	R5	Nonempty room	r ∈ Room: !r.hasPerson
Q5	R5	Mark room as nonempty	r.hasPerson=true
Q6	R5	Mark room as empty	r.hasPerson=false
Q7-Q9	R6	Q4-Q6 with office instead of room	

Table 1. The query and command repository: Queries are constructed from the schema entities and operate on the global context

ments. Fig. 9 presents the resulting context schema. In devising the context schema, we adopt the UML class diagram notation. Other languages may be used as well.

Context Population Once the context schema is designed, we populate it with the initial predefined data. That is, associating the actual rooms to buildings, as well as, binding the devices to the specific rooms. This can be done visually using some kind of object diagram (instantiating the class diagram) or by populating a context database.

Context Composition We now turn to define the context query and command repository. The repository contains 'select' queries and 'update' commands for querying and changing the context data. Table 1 lists the queries for our example.

Context-based Behavioral Specification In information systems, the output of database queries is often used for reports. Here, we propose to use queries' output to dynamically bind contexts to LSC charts. In our use case, e.g., the LSC that ensures that the room's light will not turn off during an emergency (Fig. 4) binds to an Emergency instance and to all Room instances.

With these steps, we outlined our vision of how contextual LSC can be used for allowing richer specifications for IoT systems. With the right tool support, this methodology allows for better involvement of stakeholders in the specification of IoT behaviors, by providing them with the subject domain vocabulary needed for efficient communication.

As development evolves, new requirements may arise. In our example these include:

Physical: **R8**) Smart speakers are installed in all rooms (in addition to R2 devices); **R9**) Workers can be identified in the system (e.g., by smartphone), while visitors cannot.

Behavioral: **R10**) If a worker enters a room, announce her name in the room's speaker; **R11**) During an emergency, turn on all lights.

To cope with these requirements, we can add object types that represent new aspects of the context. To the schema, we add a Smart Speaker data type (similarly to the other devices) (R8), and a Worker data type, both associated with Room (see Fig. 10). To the queries repository we add commands for marking and un-marking that a worker has entered a room (R10) (similar to Q5-Q6). For getting a view of all workers that are inside a room (R9), we add the "Worker in a room" query, defined as "$w \in Worker,\ r \in Room$: $r.workers.includes(w)$)". Finally, we add the following LSC charts: two for detecting a worker that has entered/left a room and executing the appropriate marking/unmarking command (R9, R10); one for turning the lights on during an emergency (R11); and one for announcing the worker name (Fig. 11) (R10).

Thanks to the incrementality feature of behavioral programming paradigm, no changes to the previous specification are needed, we only added elements to the schema, to the query and command repository, and to the list LSC charts.

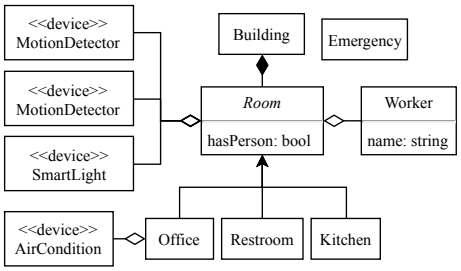


Fig. 10. Schema for the new requirements

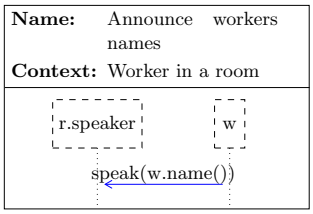


Fig. 11. Announce workers names

6 Conclusions and Future Work

We identified the needs and proposed a modeling language towards more robust and manageable specifications for the behavior of IoT. We argued that IFTTT rules may become unwieldy as the complexity grows, and showed how our language allows for consolidation of several rules into a single specification object. We also developed a proof-of-concept tool that allows us to produce and execute the diagrams in this paper. With this experience, we proposed a methodology for specifying the behavior of IoT.

In future work, plan to quantify and validate the advantages of the proposed language in comparison to existing ones, like IFTTT, via an empirical study in which subjects will specify an IoT system in multiple ways. We also plan to integrate the language and methodology in a holistic development environment.

References

1. G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *International symposium on handheld and ubiquitous computing*, pages 304–307. Springer, 1999.
2. L. Da Xu, W. He, and S. Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, 10(4):2233–2243, 2014.
3. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des.*, 19(1):45–80, 1987.
4. T. Eterovic, E. Kaljic, D. Donko, A. Salihbegovic, and S. Ribic. An IoT visual domain specific modeling lang. based on UML. In *ICAT XXV*, pages 1–5, 2015.
5. D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: towards a comprehensive tool for scenario based programming. In *ASE*, New York, New York, USA, sep 2010. ACM Press.
6. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer Science & Business Media, 2003.
7. D. Harel, A. Marron, and G. Weiss. Behavioral programming. *CACM*, 55(7):90, 2012.
8. D. Harel and A. Pnueli. On the Development of Reactive Systems. *Logics Model. Concurr. Syst.*, pages 477 – 498, 1985.
9. C. Perera, C. H. Liu, and S. Jayawardena. The Emerging Internet of Things Marketplace from an Industrial Perspective: A Survey. *IEEE Tr. Emerg. Topics. Comp.*, 3(4):585, 2015.
10. C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE Commun. Surv. Tutorials*, 16(1):414–454, 2014.
11. Samsung. ARTIK Cloud Homepage.
12. H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé, editors. *Vision and Challenges for Realising the Internet of Things*. Publications Office of the EU, Luxembourg, 2010.
13. O. Vermesan, P. Friess, P. Guillemin, S. Gusmeroli, H. Sundmaeker, et al. IoT strategic research roadmap. *IoT-Global Technological and Societal Trends*, 1(2011):9–52, 2011.
14. B. Wasik. In the Programmable World, All Our Objects Will Act as One. *Wired*, 2013.
15. C. Wootton. *Beginning Samsung ARTIK*. Springer, 2016.